



DEMYSTIFYING CSP FOR MODERN APPLICATIONS

DR. PHILIPPE DE RYCK

<https://PragmaticWebSecurity.com>

Content Security Policy 1.0

Content Security

W3C Working Draft, 11 February 2015

▼ More details about this document

This version:

<https://www.w3.org/TR/2015/WD-CSP-20150211/>

Latest published version:

<https://www.w3.org/TR/CSP/>

Editor's Draft:

<https://w3c.github.io/webappsec-csp/>

Previous Versions:

<https://www.w3.org/TR/2015/WD-CSP-20150211/>

History:

<https://www.w3.org/standards/history/CSP/>

Feedback:

public-webappsec@w3.org

[Github](#)

Editors:

[Mike West](#) (Google)

[Antonio Sartori](#) (Google)

Content Security Policy Level 2

Content Security

W3C Working Draft, 11 February 2016

▼ More details about this document

This version:

<https://www.w3.org/TR/2016/WD-CSP2-20160211/>

Latest published version:

<https://www.w3.org/TR/CSP3/>

Editor's Draft:

<https://w3c.github.io/webappsec-csp/>

Previous Versions:

<https://www.w3.org/TR/2016/WD-CSP2-20160211/>

History:

<https://www.w3.org/standards/history/CSP2/>

Feedback:

public-webappsec@w3.org with

[Github](#)

Editors:

[Mike West](#) (Google Inc.)

[Antonio Sartori](#) (Google Inc.)

Content Security Policy Level 3

W3C Working Draft, 11 February 2026

▼ More details about this document

This version:

<https://www.w3.org/TR/2026/WD-CSP3-20260211/>

Latest published version:

<https://www.w3.org/TR/CSP3/>

Editor's Draft:

<https://w3c.github.io/webappsec-csp/>

Previous Versions:

<https://www.w3.org/TR/2026/WD-CSP3-20260206/>

History:

<https://www.w3.org/standards/history/CSP3/>

Feedback:

public-webappsec@w3.org with subject line “[CSP3] ... message topic ...” ([archives](#))

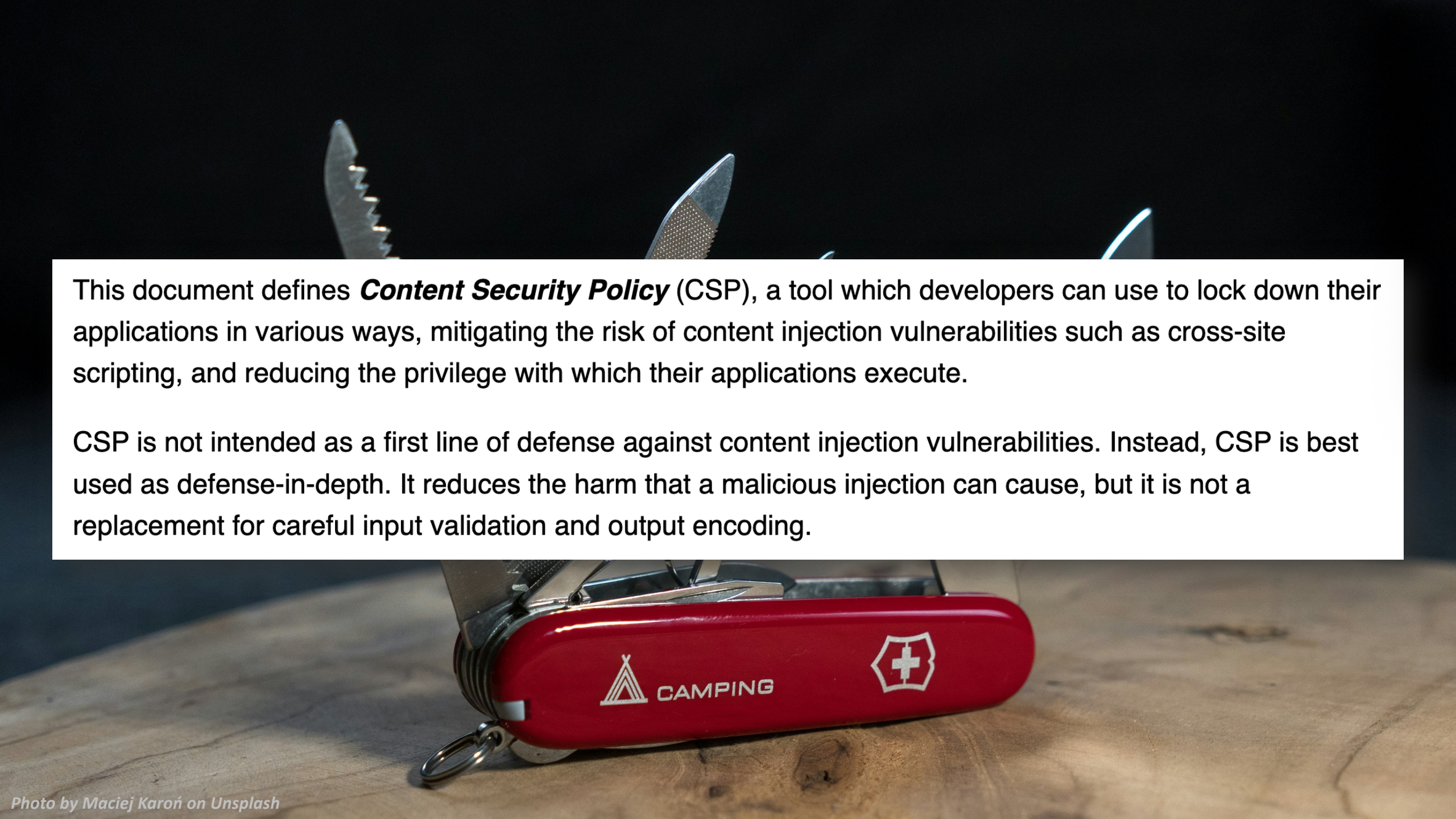
[Github](#)

Editors:

[Mike West](#) (Google Inc.)

[Antonio Sartori](#) (Google Inc.)

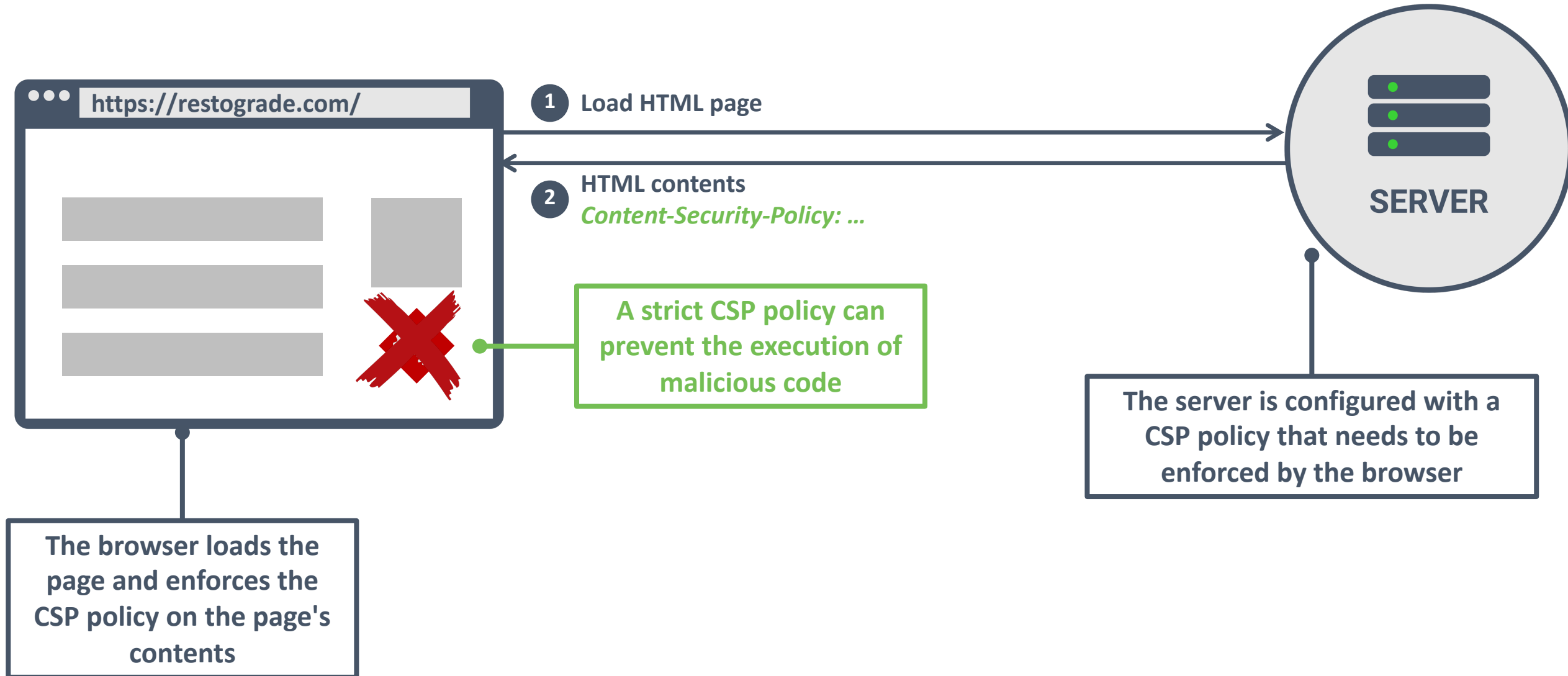




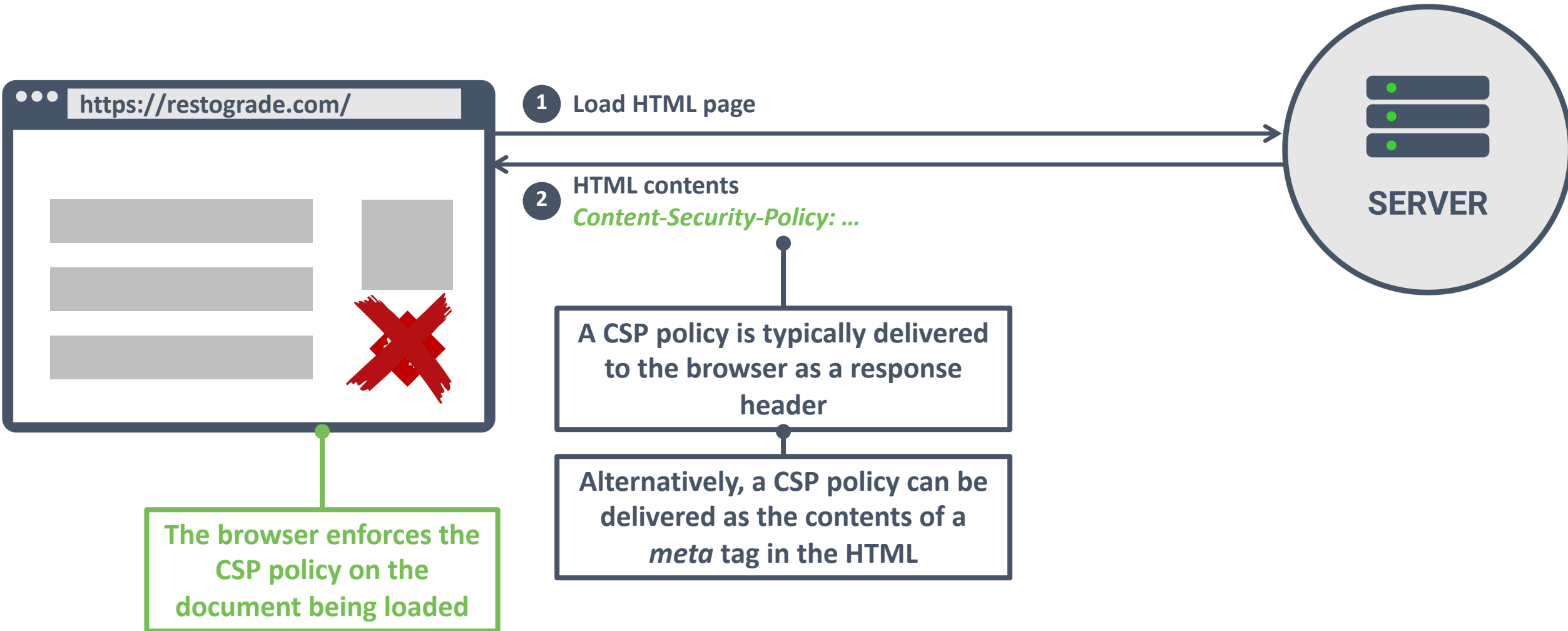
This document defines **Content Security Policy** (CSP), a tool which developers can use to lock down their applications in various ways, mitigating the risk of content injection vulnerabilities such as cross-site scripting, and reducing the privilege with which their applications execute.

CSP is not intended as a first line of defense against content injection vulnerabilities. Instead, CSP is best used as defense-in-depth. It reduces the harm that a malicious injection can cause, but it is not a replacement for careful input validation and output encoding.

CSP AS A SECOND LINE OF DEFENSE AGAINST XSS



DEPLOYING CSP FOR A DOCUMENT





Preventing XSS with CSP

MITIGATING XSS WITH CSP

A CSP policy deployed by the application

1 `Content-Security-Policy: script-src 'self'`

This policy only allows the execution of script files from the application's own origin

XSS through inline code blocks

1 `<div><script>alert(1)</script></div>`

Inline code blocks are not coming from the own origin, so they are not executed

XSS through inline code

1 ``
2 `<iframe src="javascript:alert(1)">`
3 `<iframe src="data:text/html,script>alert(1)</script">`

Inline code is not coming from the own origin, so these code snippets are not executed

XSS through remote code files

1 `<div><script src="https://evil.com/hacked.js"></script></div>`

This remote code file is not coming from the own origin, so it is not executed

CSP ACTS AS A SECONDARY DEFENSE AGAINST INJECTION



A properly configured CSP policy can protect your application in case an injection vulnerability exists.

CSP is only a second line of defense. The first line of defense is proper output encoding and sanitization to mitigate injection vulnerabilities.



**What is the effect of CSP on
your application code?**

I am *Dr. Philippe De Ryck*



Founder of Pragmatic Web Security



Google Developer Expert



SecAppDev organizer

I help developers with security



Hands-on in-depth security training



Advanced online security courses



Expert security advisory services



<https://pdr.online>



There are three ways to re-enable inline code blocks with CSP

A CSP policy using 'unsafe-inline' for scripts

1 **Content-Security-Policy:** `script-src 'self' 'unsafe-inline'`

'unsafe-inline' re-enables inline script code, both legitimate code and injected code.

'unsafe-inline' was the only way to enable inline script code in CSP level 1

This hash uniquely identifies the code block below down to a space, allowing that exact code block to run, even when specified inline

A CSP policy deployed by the application

```
1 Content-Security-Policy: script-src 'sha256-Y1qZpipLn29Prju...eeCKWH4Ubm0tB9LUs='
```

A code snippet from the application containing an inline code block

```
1 <body>
2   <div>...</div>
3   <script>... doSomething() ...</script>
4 </body>
```

In CSP level 2, we can add a hash value of the exact contents of this script block, so that we can mark it as allowed to execute



Using hashes for inline code blocks



Hashes can only be used when the inline code block contains static code

This nonce is used to identify legitimate code blocks which also carry the nonce, allowing them to be executed

A CSP policy deployed by the application

```
1 Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK'
```

A code snippet from the application containing an inline code block

```
1 <script nonce="x4GACP2dm0UCK">  
2   ...  
3   inline script code  
4   ...  
5 </script>  
6
```

In CSP level 2, we can add a nonce to the policy and to a script tag, marking a specific code block as approved



Using nonces for inline code blocks



Nonces should be unpredictable, so they must be different on every page load

Content Security Policy Level 2 - REC

Usage

% of all users

Global

96.32% + 0.02% = 96.34%

Baseline Widely available across major browsers

Mitigate cross-site scripting attacks by only allowing certain sources of script, style, and other resources. CSP 2 adds hash-source, nonce-source, and five new directives

Current aligned Usage relative Date relative Filtered All

Chrome	Edge *	Safari	Firefox	Opera	IE	Chrome for Android	Safari on iOS *	Samsung Internet	Opera Mini *	Opera Mobile	UC Browser for Android	Android Browser *	Firefox
			2-30										
4-35			¹ 31-34	10-22									
⁴ 36-38	12-14		² 35	⁴ 23-25									
⁵ 39	⁶ 15-18	3.1-9.1	³ 36-44	⁵ 26			3.2-9.3						
40-144	79-144	10-26.2	45-146	27-124	6-10		10-26.2	4-28		12-12.1		2.1-4.4.4	
145	145	26.3	147	125	11	145	26.3	29	all	80	15.5	145	
146-148		26.4-TP	148-150				26.4						

NONCES AND HASHES ARE GREAT FOR INLINE CODE



By including a hash or nonce in the CSP policy, we can selectively re-enable legitimate inline code blocks.

When the browser recognizes a hash, it will ignore the 'unsafe-inline' expression.



There are three ways to load remote code files with CSP

This URL expression identifies where scripts can be included from

A CSP policy deployed by the application

```
1 Content-Security-Policy: script-src https://cdn.restograde.com
```

Note that CSP expressions are flexible. They can point to a host, to a specific file (jquery.js), or even use wildcards.

A code snippet from the application containing a remote code file

```
1 <script src="https://cdn.restograde.com/jquery.js"></script>
```

In CSP level 1, the browser checks the URL of the script against the *script-src* directive

ADDING CDNS TO CSP



CSP Is Dead, Long Live CSP! On the Insecurity of Whitelists and the Future of Content Security Policy

Lukas Weichselbaum
Google Inc.
lwe@google.com

Michele Spagnuolo
Google Inc.
mikispag@google.com

Sebastian Lekies
Google Inc.
slekies@google.com

Artur Janc
Google Inc.
aaj@google.com

ABSTRACT

Content Security Policy is a web platform mechanism designed to mitigate cross-site scripting (XSS), the top security vulnerability in modern web applications [24]. In this paper, we take a closer look at the practical benefits of adopting

1. INTRODUCTION

Cross-site scripting – the ability to inject attacker-controlled scripts into the context of a web application – is arguably the most notorious web vulnerability. Since the first formal reference to XSS in a CERT advisory in 2000

<https://research.google/pubs/pub45542/>

In total, we find that 94.68% of policies that attempt to limit script execution are ineffective

COMMON CSP BYPASSES

Approving a host that hosts a JSONP endpoint. Examples include Doubleclick ads and integrated timeline code from Twitter.

Approving a host that hosts a version of AngularJS 1.x. This legacy version of Angular includes a template compiler, allowing the attacker to leverage Angular to transform template code ({{{...}}}) into executable code.

Approving your own origin ('self'), which may host some of the above, or user-uploaded files.

Not blocking the loading of Flash files, allowing the attacker to include a vulnerable Flash file and trick it into executing JavaScript code



CSP Evaluator

CSP Evaluator allows developers and security experts to check if a Content Security Policy (CSP) serves as a strong mitigation against [cross-site scripting attacks](#). It assists with the process of reviewing CSP policies, which is usually a manual task, and helps identify subtle CSP bypasses which undermine the value of a policy. CSP Evaluator checks are based on a [large-scale study](#) and are aimed to help developers to harden their CSP and improve the security of their applications. This tool (also available as a [Chrome extension](#)) is provided only for the convenience of developers and Google provides no guarantees or warranties for this tool.

Content Security Policy

[Sample unsafe policy](#) [Sample safe policy](#)

Paste CSP or URL (starting with [http://](#) or [https://](#)) here.

CSP Version 3 (nonce based + backward compatibility checks) ?

CHECK CSP

❗ https://*.linkedin.com	www.linkedin.com is known to host JSONP endpoints which allow to bypass this CSP.
❗ www.google.com	www.google.com is known to host JSONP endpoints which allow to bypass this CSP.
🔍 www.google-analytics.com	No bypass found; make sure that this URL doesn't serve JSONP replies or Angular libraries.
🔍 *.googletagmanager.com	No bypass found; make sure that this URL doesn't serve JSONP replies or Angular libraries.
❗ www.googleadservices.com	www.googleadservices.com is known to host Angular libraries which allow to bypass this CSP.
❗ googleads.g.doubleclick.net	googleads.g.doubleclick.net is known to host JSONP endpoints which allow to bypass this CSP.
❗ https://*.doubleclick.net/	googleads.g.doubleclick.net is known to host JSONP endpoints which allow to bypass this CSP.
🔍 ssl.google-analytics.com	No bypass found; make sure that this URL doesn't serve JSONP replies or Angular libraries.
❗ https://www.gstatic.com	www.gstatic.com is known to host Angular libraries which allow to bypass this CSP.

URLS ARE NOT RECOMMENDED FOR SCRIPT-SRC



Many CSP policies relying on URLs are overly permissive, allowing an attacker to easily bypass the CSP policy when abusing an injection vulnerability.



There are ~~three~~ two ways to load remote code files with CSP

This nonce in the policy is used to identify approved script tags (inline or remote)

A CSP policy deployed by the application

```
1 Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK'
```

Note that the host (cdn.restograde.com) is not explicitly listed in the policy. The nonce suffices to approve a remote code file

A code snippet from the application containing a remote code file with a nonce

```
1 <script nonce="x4GACP2dm0UCK" src="https://cdn.restograde.com/jquery.js"></script>
```

In CSP level 2, we can use a nonce to approve remote code files as well (the nonce is independent of the file contents, making this possible in CSP level 2)

This hash uniquely identifies the code file, allowing that exact code file to run. This hash is identical to the *Subresource Integrity* hash

A CSP policy deployed by the application

```
1 Content-Security-Policy: script-src 'sha256-Y1qZpipLn29Prju...eeCKWH4Ubm0tB9LUs='
```

A code snippet from the application containing a remote code file

```
1 <script src="https://cdn.restograde.com/jquery.js"  
2     integrity="sha256-Y1qZpipLn29Prju...eeCKWH4Ubm0tB9LUs="  
3     crossorigin="anonymous">  
4 </script>
```

CSP's remote hashes require the use of *Subresource Integrity* on the script file



Using hashes for remote code files



Universal browser support for remote hashes only became available in August 2023

headers HTTP header: Content-Security-Policy: script-src: External scripts with hash



Usage

% of all users



Global

95.46%

Current aligned

Usage relative

Date relative

Filtered

All



Chrome	Edge *	Safari	Firefox	Opera	IE ! *	Chrome for Android	Safari on iOS *	Samsung Internet
4-58	12-18	3.1-15.5	2-115	10-45			3.2-15.5	4-6.4
59-144	79-144	15.6-26.2	116-146	46-124	6-10		15.6-26.2	7.2-28
145	145	26.3	147	125	11	145	26.3	29
146-148		26.4-TP	148-150				26.4	

LOAD REMOTE CODE FILES WITH HASHES OR NONCES



Nonces are widely supported for loading remote code files and are considered more secure than a URL expression.

Hashes also work quite well for loading remote code files, but legacy browsers may lack support.



What if one script needs to load more scripts?

Scripts are approved with a nonce-only policy

A nonce-only CSP policy deployed by the application

```
1 Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK'
```

A code snippet from the application which applies nonce propagation to load another code file

```
1 <script nonce="x4GACP2dm0UCK">
2   var s = document.createElement("script");
3   s.setAttribute("nonce", document.currentScript.nonce);
4   s.src = "https://trusted.example.com/myscript.js";
5   document.body.appendChild(s);
6 </script>
```

The nonce marking a code block as valid is propagated on a script tag to load a remote script file

The browser does not expose the nonce in the DOM, but makes it programmatically available to running code at `document.currentScript.nonce`



If you are building a JS library, it is a best practice to support nonce propagation when you load additional code

The *strict-dynamic* expression enables automatic trust propagation in the browser

A nonce-only CSP policy deployed by the application

```
1 Content-Security-Policy: script-src 'nonce-x4GACP2dm0UCK' 'strict-dynamic'
```

A code snippet from the application which applies nonce propagation to load another code file

```
1 <script nonce="x4GACP2dm0UCK">  
2   var s = document.createElement("script");  
3   s.src = "https://trusted.example.com/myscript.js";  
4   document.body.appendChild(s);  
5 </script>
```

A script loaded with a hash or a nonce is considered a secure starting point for automatic trust propagation

Because of *strict-dynamic*, the browser allows an approved script to load additional scripts through the proper DOM APIs



When the browser sees *strict-dynamic*, it will ignore URL expressions, including *'self'*



Trust propagation in action

TRUST PROPAGATION SIMPLIFIES CSP POLICIES

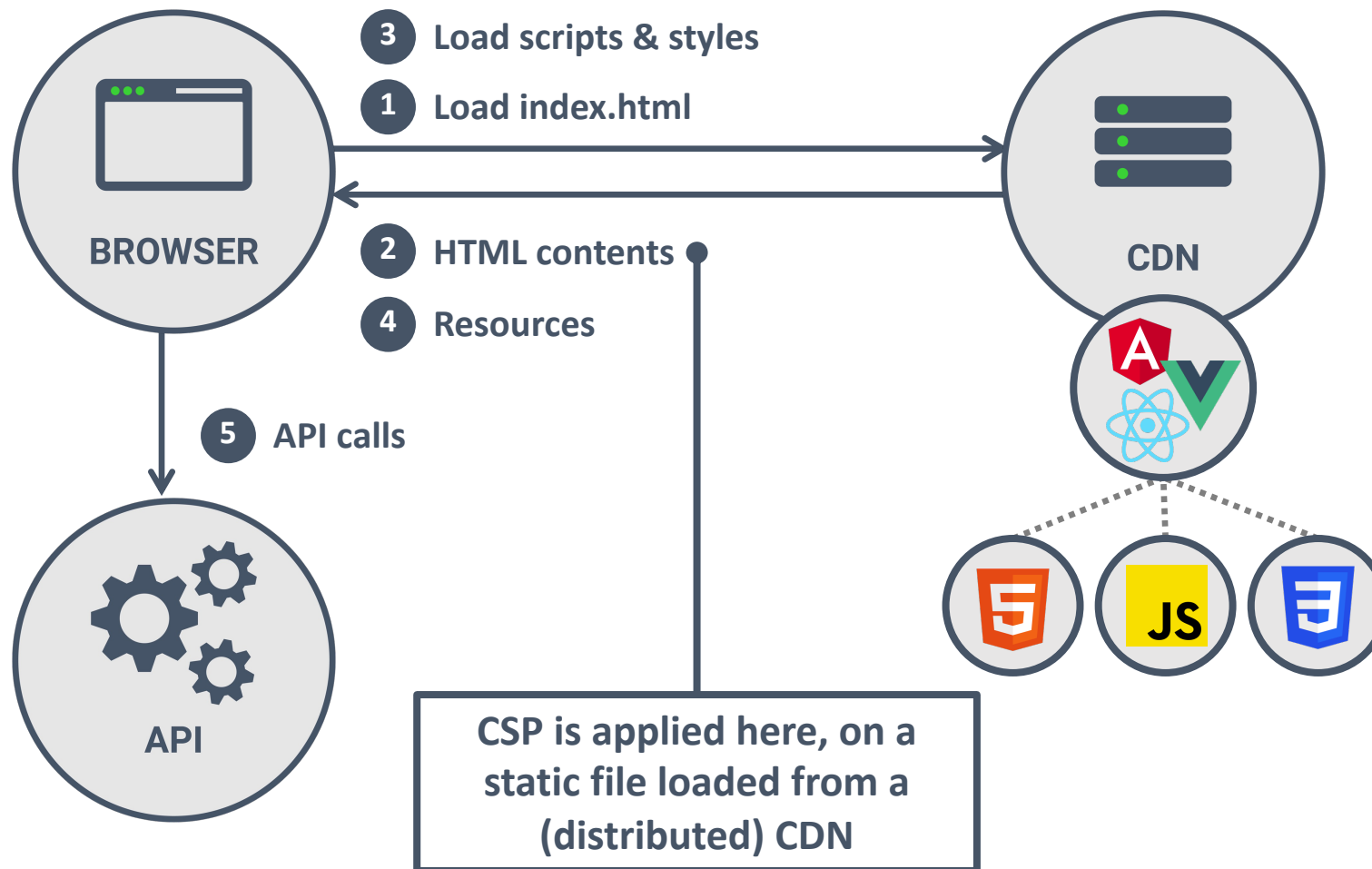


Using manual nonce propagation or automatic trust propagation allows approved scripts to load additional code.

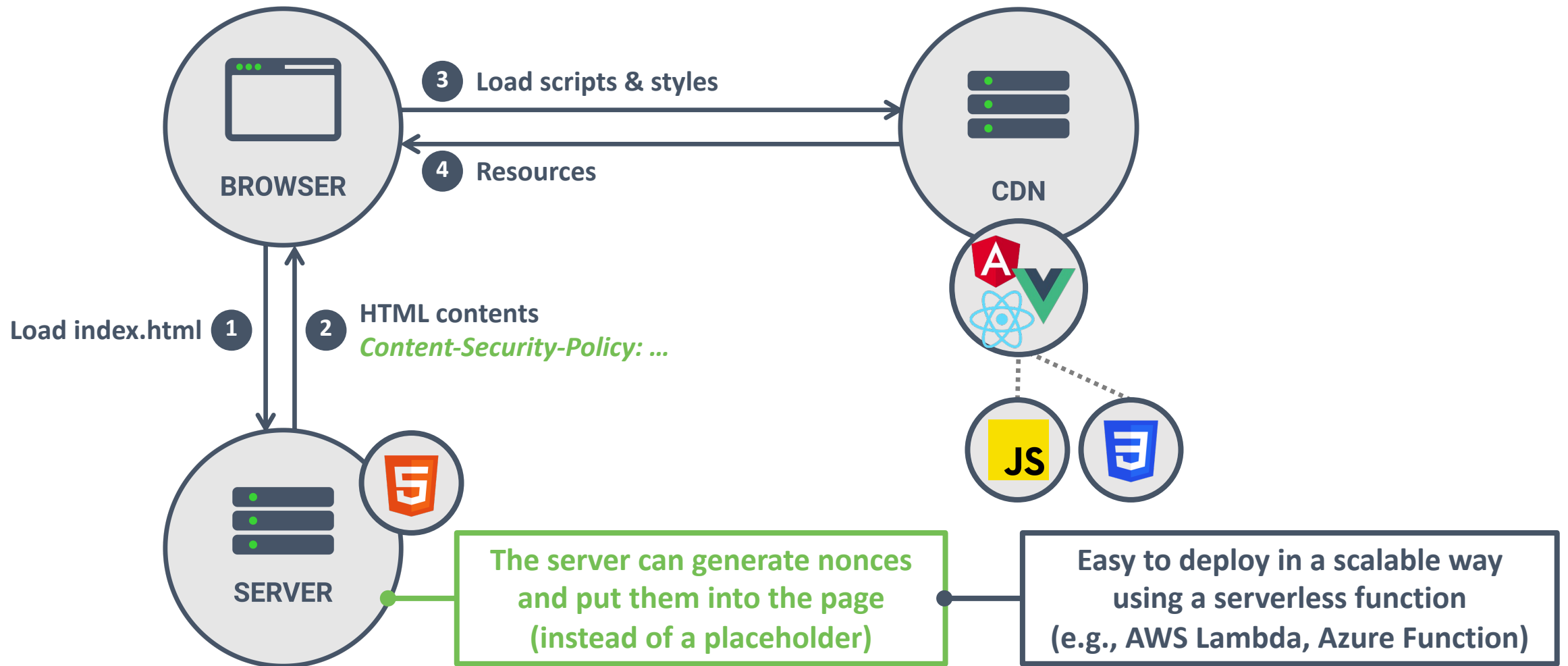
This enables a crucial feature without bloating the policy with all kinds of hosts and files.

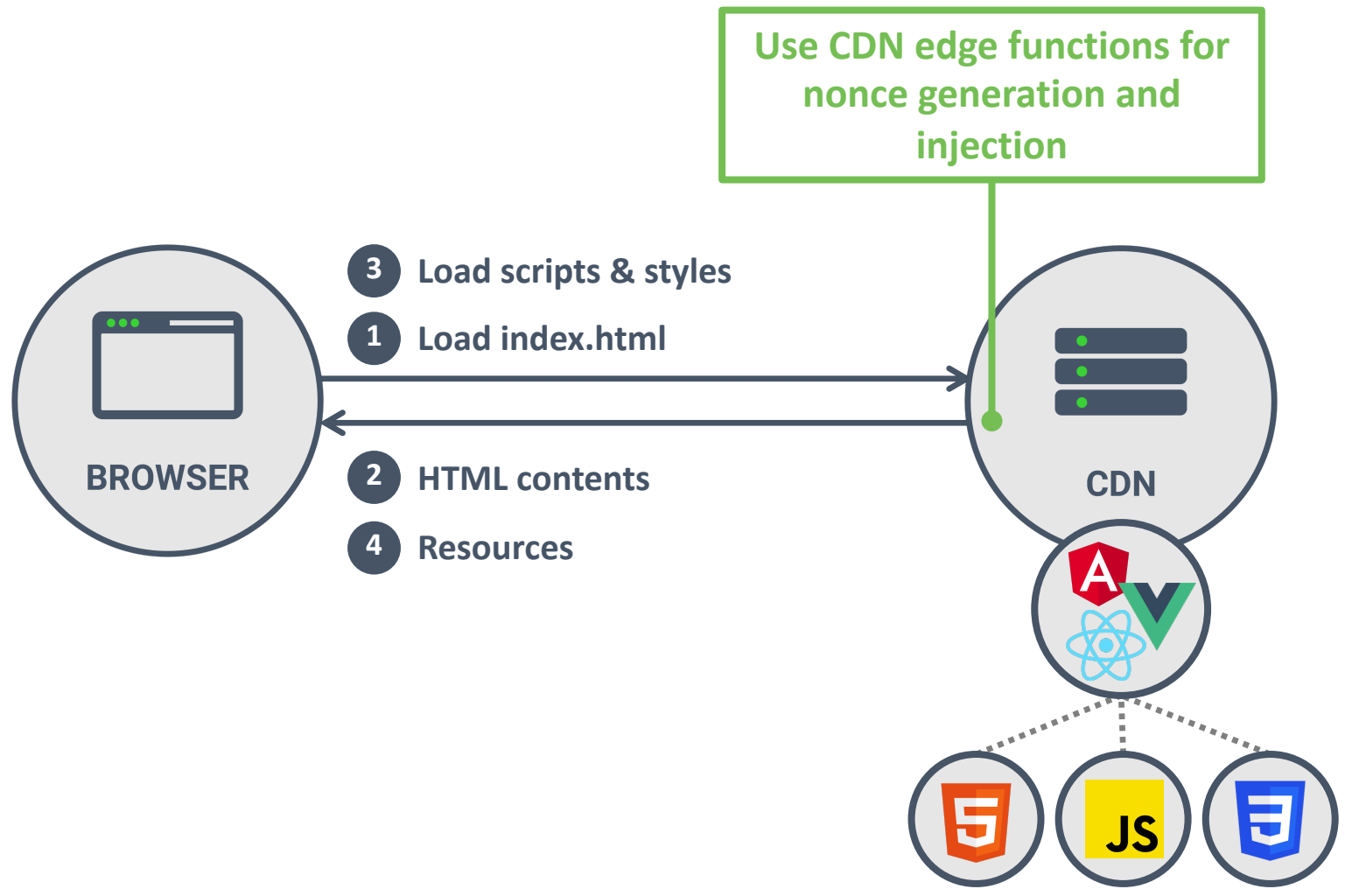


How do we apply this CSP knowledge to modern web frontends?



DYNAMICALLY SERVING SPAs






Content Security Policy

Installed

Every request generates a unique nonce for the `script-src` directive of your Content Security Policy (CSP).



Security

Developed by  Netlify

This extension is maintained by Netlify, and is verified to meet best practices and platform standards.

◆ On this page

Extension details

Danger zone

Extension details

Use a [nonce](#) ↗ for the `script-src` directive of your Content Security Policy (CSP) to help prevent [cross-site scripting \(XSS\)](#) ↗ attacks.

This extension deploys an edge function that adds a response header and transforms the HTML response body to contain a unique nonce on every request, along with an optional function to log CSP violations.

Scripts that do not contain a matching `nonce` attribute, or that were not created from a trusted script (see [strict-dynamic](#) ↗), will not be allowed to run.

You can use this extension whether or not your site already has a CSP in place. If your site already has a CSP, the nonce will merge with your existing directives.

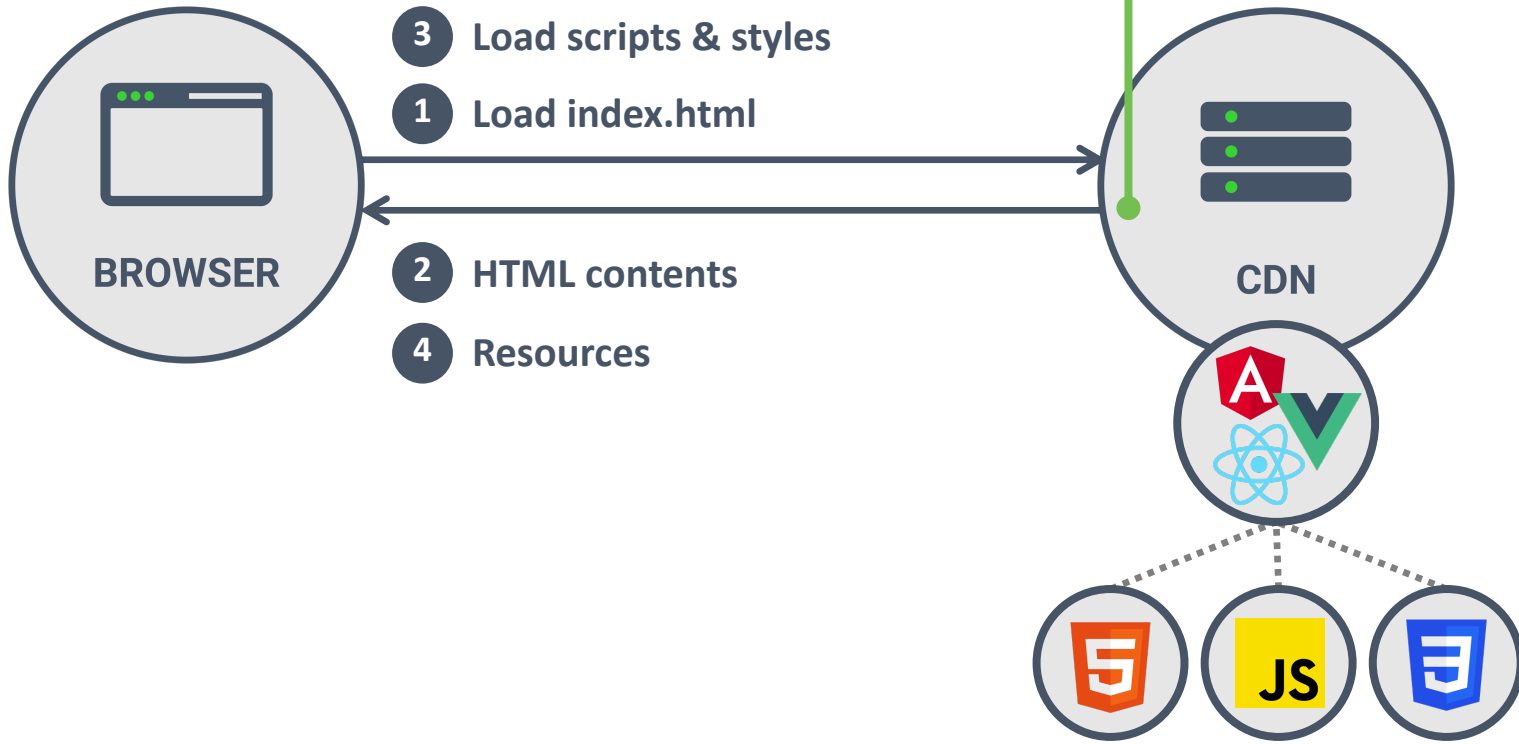
🧩 This extension is installed and configured in the Netlify UI. If you prefer a configuration-as-code approach, check out the [@netlify/plugin-csp-nonce](#) ↗ npm package.

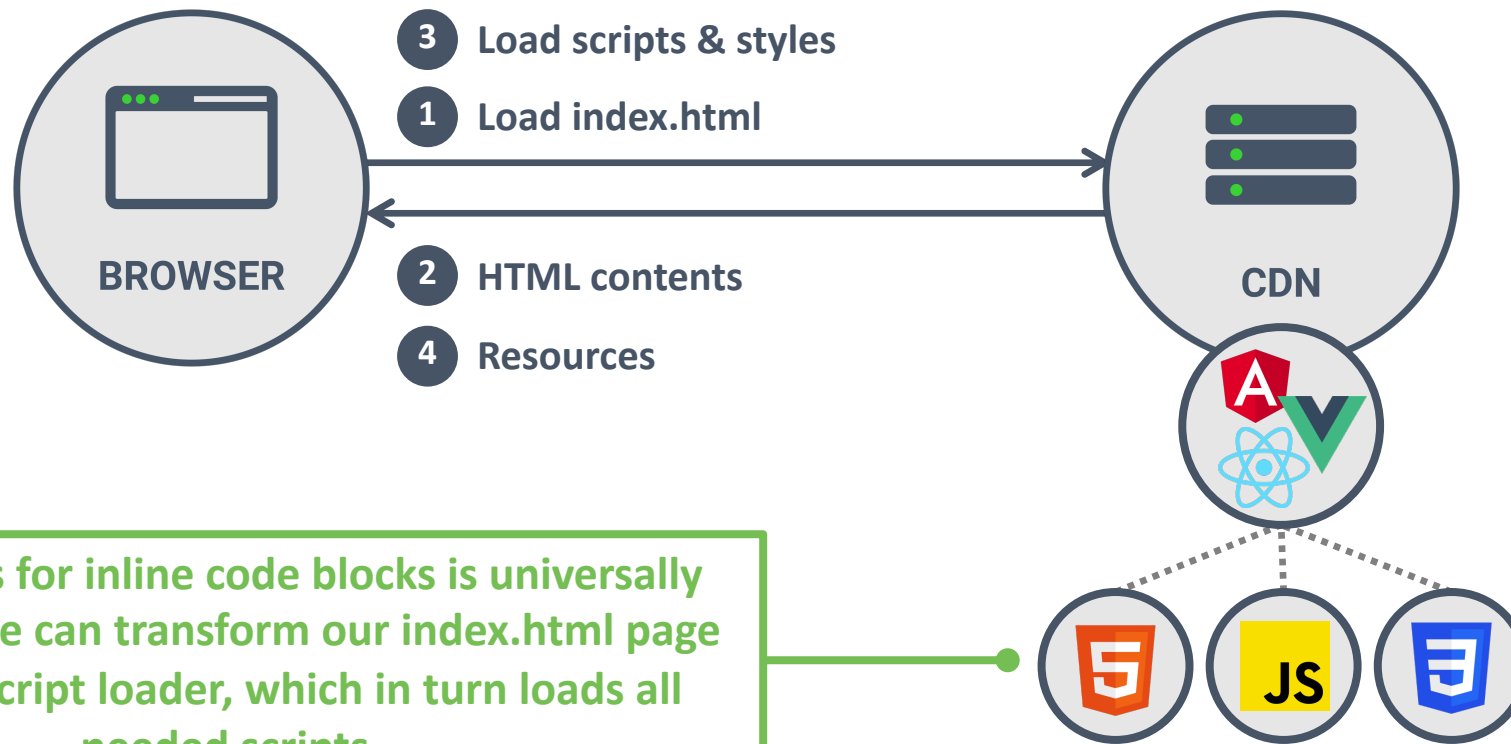


Once the first scripts are loaded with a nonce, trust propagation allows the application to (lazy) load additional code

Remote hashes may not be supported widely enough for your liking

Remote hashes are easy to use, since most build environments already enable Subresource Integrity





Using hashes for inline code blocks is universally supported. We can transform our index.html page to inject a script loader, which in turn loads all needed scripts

ENABLING A CSP SCRIPT LOADER IN ANGULAR

Enabling CSP through build options in angular.json

```
1 "security": {  
2   "autoCsp": true  
3 }
```

Setting *autoCsp* to true enables a build step in Angular 19+ to transform the *index.html* page to support CSP's *'strict-dynamic'*

The generated code in index.html

```
1 <head><meta http-equiv="Content-Security-Policy" content="script-src 'strict-dynamic' 'sha256-  
2 uv6sWg...N/QREkJ1w=' https: 'unsafe-inline';object-src 'none';base-uri 'self';">  
3 <script>  
4 var scripts = [  
5   ['polyfills-FFHMD2TL.js', 'module', false, false], ['main-YDH4PR7Z.js', 'module', false, false]  
6 ];  
7  
8 scripts.forEach(function(scriptUrl) {  
9   var s = document.createElement('script');  
10  s.src = scriptUrl[0];  
11  s.type = scriptUrl[1];  
12  s.async = !!scriptUrl[2];  
13  s.defer = !!scriptUrl[3];  
14  document.lastElementChild.appendChild(s);  
15 });  
16 </script>
```



Once the first scripts are loaded with a hash, *strict-dynamic* allows the application to (lazy) load additional code

CSP IS QUITE COMPATIBLE WITH MODERN FRONTENDS



Dynamic nonce injection, remote hashes, and inline script loaders offer concrete strategies for configuring a secure CSP policy without URL expressions.

A backwards compatible CSP policy

```
1 Content-Security-Policy:  
2   script-src 'nonce-+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3   'unsafe-inline' http: https:
```

*Unsafe-inline is ignored
when the browser observes
the use of a hash or a nonce*

*URL-based entries are
ignored when the browser
observes *strict-dynamic**

A backwards compatible CSP policy as observed by modern browsers

```
1 Content-Security-Policy:  
2   script-src 'nonce+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3   'unsafe-inline' http: https:
```



Safari 15.4 added support for strict-dynamic

A backwards compatible CSP policy as observed by old browsers (CSP Level 2)

```
1 Content-Security-Policy:  
2   script-src 'nonce+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3   'unsafe-inline' http: https:
```

These policies add no security, but they also don't break the application

A backwards compatible CSP policy as observed by IE (CSP level 1)

```
1 Content-Security-Policy:  
2   script-src 'nonce+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3   'unsafe-inline' http: https:
```

The application is not more vulnerable than without CSP, and 95+% of users benefits from CSP's protections

CSP IS DESIGNED TO BE BACKWARDS COMPATIBLE



CSP policies can be constructed so that they do not break the application on older browsers.

Understanding the security of a CSP policy requires careful consideration of enabled features and common browser versions of your users.


More awesome CSP features

```
1 Content-Security-Policy:  
2     script-src 'nonce-+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3         'unsafe-inline' http: https:  
4     report-uri ...
```

***Report-uri* enables the reporting of violations. The browser will send a report to the specified endpoint when it observes a violation of the CSP policy**

An example of the body of a CSP report request

```
1  {
2    "csp-report": {
3      "document-uri": "https://cspdemo.restograde.com/",
4      "referrer": "",
5      "violated-directive": "script-src-elem",
6      "effective-directive": "script-src-elem",
7      "original-policy": "script-src 'self'; report-uri https://cspdemo.restograde.com/report",
8      "disposition": "report",
9      "blocked-uri": "https://maliciousfood.com/js/evil.js",
10     "status-code": 200,
11     "script-sample": ""
12   }
13 }
```



CSP reports give detailed information about the violation, the policy that was in action, and where the violation occurred



CSP reporting in action

Try out CSP with a report-only version, with zero risk of breaking things

```
1 Content-Security-Policy-Report-Only:  
2     script-src 'nonce-+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3         'unsafe-inline' http: https:  
4     object-src 'none'  
5     base-uri 'self'  
6     report-uri ...
```

CSP REPORTING GIVES YOU INSIGHTS IN VIOLATIONS



CSP was one of the first browser security policies to support reporting. Reports give insights in client-side violations, allowing you to respond to issues or attacks.

Report-Only mode is the perfect tool to test a CSP policy before switching it on as a blocking policy.

More awesome CSP features

```
1 Content-Security-Policy:  
2   script-src 'nonce+wb8eWh0/5ihwKk20YeWRg' 'strict-dynamic'  
3           'unsafe-inline' http: https:  
4   report-uri ...  
5   object-src 'none'  
6   base-uri 'self'
```

Object-src controls embedded content and can typically be set to *'none'* (Flash is dead!)

base-uri controls valid values for the *base* tag. Setting it to *'self'* prevents base jumping attacks with injected *base* tags



CSP allows you to control other types of resources and outgoing requests ...

style-src

Controls the loading of style sheets. Supports *'unsafe-inline'*, hashes, and nonces.

img-src

Controls the loading of images. Supports URL expressions including wildcards.

font-src

Controls the loading of fonts. Supports URL expressions including wildcards.

media-src

Controls the loading of media files. Supports URL expressions including wildcards.

child-src

Controls the loading of child iframes. Supports URL expressions including wildcards.

connect-src

Controls Fetch and WebSocket connections. Supports URL expressions including wildcards.

worker-src

Controls where service workers can be loaded from.

default-src

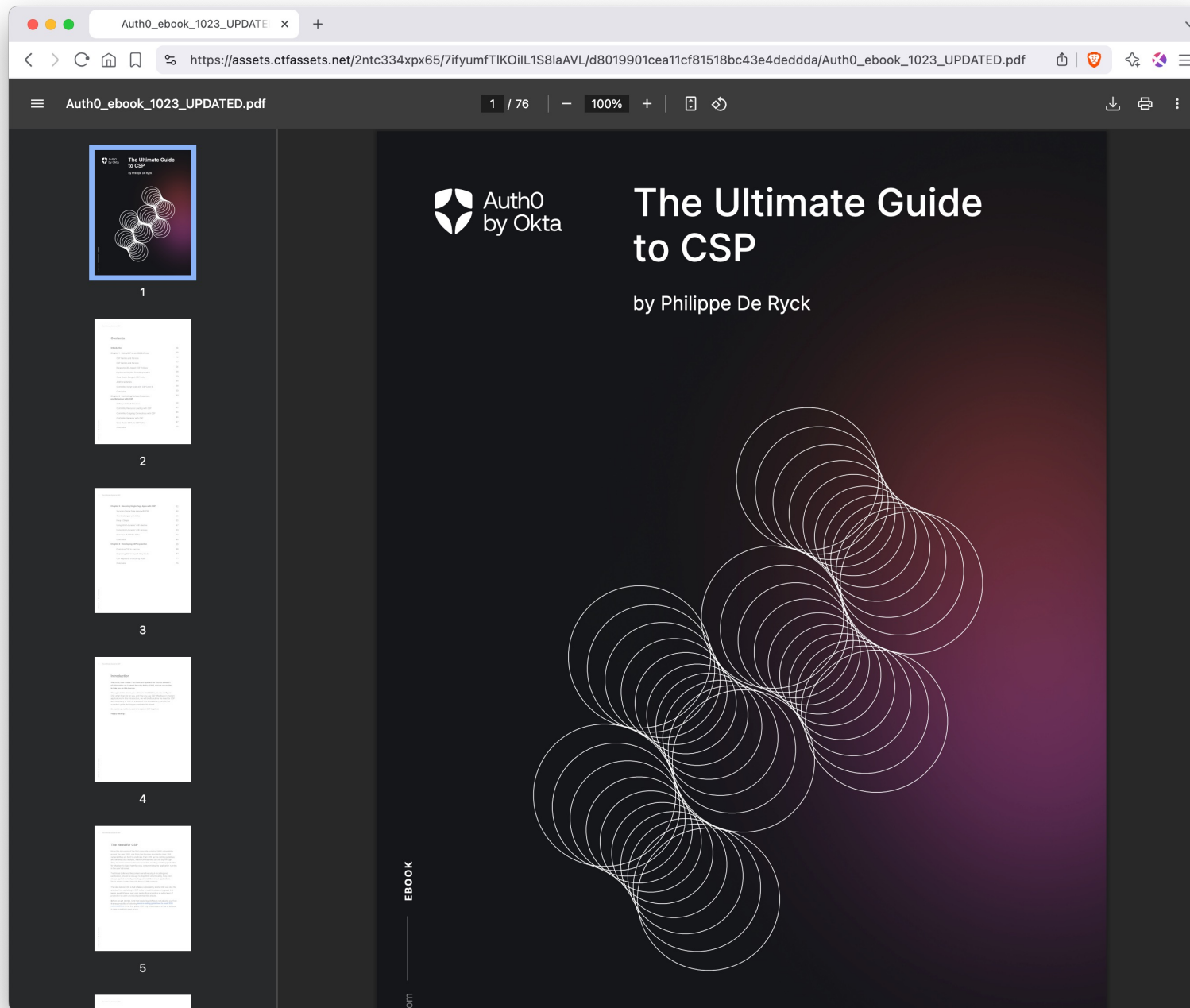
A fallback used when there is no more specific directive. Typically set to 'none' or 'self'

form-action

Controls where forms can be submitted to.

A fine-grained CSP policy can result in a fragile and frustration experience, while adding little security

You do not need fine-grained control to benefit from CSP, so be careful before going down this road



https://assets.ctfassets.net/2ntc334xpx65/7ifyumfTIKOiIL1S8laAVL/d8019901cea11cf81518bc43e4deddda/Auth0_ebook_1023_UPDATED.pdf

KEY TAKEAWAYS

1 Modern CSP policies rely on hashes, nonces, and trust propagation

2 Modern frontends work well with CSP, but may require some work

3 CSP is only a secondary defense, so you still need to fix your apps!



Thank you!

**Need training or security guidance?
Reach out to discuss how I can help**

<https://pragmaticwebsecurity.com>